Занятие 10:
Lambda и Stream API

Лямбда-выражения

Лямбда-выражения это попытка добавить в полностью объектно-ориентированный язык функциональность. Функции это некие операции, которые можно присваивать переменным, передавать методам через аргументы и т. д.

Упрощенно лямбда-выражение можно определить как анонимный метод без определения (. Синтаксис вызова лямбда-выражения:

```
(аргументы) -> (тело)
```

Java компилирует лямбда-выражения с преобразованием и в приватные методы класса.

Лямбда-выражения хорошо подходят для определения имплементаций функциональных интерфейсов.

Синтаксис лямбда-выражений

- ✓ Lambda-выражения могут иметь от 0 и более входных параметров.
- ✓ Тип параметров может быть получен из контекста. Например (int a) то же самое что (a)
- ✓ Параметры заключаются в круглые скобки и разделяются запятыми. Например (a, b) или (int a, int b) или (String a, int b, float c)
- ✓ В отсутствие параметров можно использовать круглые скобки. Например () -> 42
- ✓ Если параметр один и его тип не указывается, скобки можно опустить. Пример: a -> return a*a
- ✓ Тело Lambda-выражения может содержать от 0 и более выражений.
- ✓ Если тело состоит из одного оператора, возвращаемое значение можно указывать без ключевого слова return.
- ✓ В других случаях фигурные скобки обязательны, а в конце надо явно использовать ключевое слово return (иначе лямбда будет возвращать void).

Функциональные интерфейсы

Функциональные интерфейсы это интерфейсы, у которых существует единственный метод. Примеры функциональных интерфейсов: Runnable, ActionListener

Чтобы точно определить интерфейс как функциональный, добавлена аннотация @FunctionalInterface, работающая по принципу @Override. Она обозначит замысел и не даст определить второй абстрактный метод в интерфейсе.

- ✓ Function<T, R> - интерфейс, с помощью которого реализуется функция, получающая на ввод экземпляр класса T и возвращающая на выходе экземпляр класса R.
- ✓ Predicate<T> - на ввод - T, возвращает результат типа boolean.
- ✓ Consumer<T> - на ввод - T, производит некое действие и ничего не возвращает.
- ✓ Supplier<T> - ничего не принимает на ввод, возвращает T
- ✓ BinaryOperator<T> - на ввод - два экземпляра T, возвращает один T

Stream API

Stream API - механизм, дающий возможность работать с источниками данных с помощью функционального подхода.

Источниками данных могут быть коллекции, или методы, поставляющие данные.

К данным применяются операторы.

Таким образом, Stream API предоставляет удобный и быстрый механизм обработки данных и большое количество инструментов, позволяющих не заботиться о деталях реализации.

Получение стримов

Из коллекции:

```
Stream<String> streamFromCollection = collection.stream();
```

Из значений:

```
Stream<String> streamFromValues = Stream.of("a1", "a2", "a3");
```

Из массива:

```
Stream<String> streamFromArrays = Arrays.stream(array);
```

Из файла (строка файла это отдельный элемент стрима)

```
Stream<String> streamFromFiles = Files.lines(Paths.get("file.txt")
```

Из строки (каждый символ это отдельный элемент стрима)

```
IntStream streamFromString = "123".chars()
```

Получение стримов

С помощью Stream.builder()

```
Stream.builder().add("a1").add("a2").add("a3").build()
```

Создание параллельного стрима

```
Stream<String> stream = collection.parallelStream();
```

Бесконечный стрим из метода Stream.iterate()

```
Stream<Integer> streamFromIterate = Stream.iterate(1, n -> n + 1)
```

Бесконечный стрим из метода Stream.generate()

```
Stream<String> streamFromGenerate = Stream.generate(() -> "a1")
```

Методы работы

Существует два вида методов работы:

- ✓ Конвейерные
- ✓ Терминальные

Конвейерные методы возвращают другой стрим, к которому можно применить следующий конвейерный, или терминальный метод.

Терминальные возвращают новый объект

Конвейерные методы

filter	Отфильтровывает записи, возвращает только записи, соответствующие условию	collection.stream().filter(«a1»::equals).count()
skip	Позволяет пропустить N первых элементов	collection.stream().skip(collection.size() — 1).findFirst().orElse(«1»)
distinct	Возвращает стрим без дубликатов (для метода equals)	collection.stream().distinct().collect(Collectors.toList())
map	Преобразует каждый элемент стрима	collection.stream().map((s) -> s + "_1").collect(Collectors.toList())
peek	Возвращает тот же стрим, но применяет функцию к каждому элементу стрима	collection.stream().map(String::toUpperCase).peek((e) -> System.out.print("," + e)).
		collect(Collectors.toList())

Конвейерные методы

limit	Позволяет ограничить выборку определенным количеством первых элементов	collection.stream().limit(2).collect(Collectors.toList())
sorted	Позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator	collection.stream().sorted().collect(Collectors.toList())
mapToInt,	Аналог map, но возвращает числовой стрим (то есть стрим из числовых примитивов)	collection.stream().mapToInt((s) -> Integer.parseInt(s)).toArray()
mapToDouble,		
mapToLong		
flatMap,	Похоже на map, но может создавать из одного элемента несколько	collection.stream().flatMap((p) -> Arrays.asList(p.split(",")).stream()).toArray(String[]::new)

Терминальные методы

findFirst	Возвращает первый элемент из стрима (возвращает Optional)	collection.stream().findFirst().orElse(«1»)
findAny	Возвращает любой подходящий элемент из стрима (возвращает Optional)	collection.stream().findAny().orElse(«1»)
collect	Представление результатов в виде коллекций и других структур данных	collection.stream().filter((s) -> s.contains(«1»)).collect(Collectors.toList())
count	Возвращает количество элементов в стриме	collection.stream().filter(«a1»::equals).count()
anyMatch	Возвращает true, если условие выполняется хотя бы для одного элемента	collection.stream().anyMatch(«a1»::equals)
noneMatch	Возвращает true, если условие не выполняется ни для одного элемента	collection.stream().noneMatch(«a8»::equals)
allMatch	Возвращает true, если условие выполняется для всех элементов	collection.stream().allMatch((s) -> s.contains(«1»))

Терминальные методы

min	Возвращает минимальный элемент, в качестве условия использует компаратор	collection.stream().min(String::compareTo).get()
max	Возвращает максимальный элемент, в качестве условия использует компаратор	collection.stream().max(String::compareTo).get()
forEach	Применяет функцию к каждому объекту стрима, порядок при параллельном выполнении не гарантируется	set.stream().forEach((p) -> p.append("_1"));
forEachOrdered	Применяет функцию к каждому объекту стрима, сохранение порядка элементов гарантирует	list.stream().forEachOrdered((p) -> p.append("_new"));
toArray	Возвращает массив значений стрима	collection.stream().map(String::toUpperCase).toArray(String[]::new);
reduce	Позволяет выполнять агрегатные функции	collection.stream().reduce((s

Параллельная обработка стримов

Любой последовательный стрим можно преобразовать в параллельный, а также наоборот
Следует помнить, что нельзя с помощью параллельных стримов обрабатывать наборы данных, работа с которыми займет продолжительное время, поскольку пул потоков используется стандартный. Во время работы могут быть приостановлены важные процессы.